## Lecture 8: Addition, Multiplication & Division

- Today's topics:
  - Signed/Unsigned
  - Addition
  - Multiplication
  - Division

• The hardware recognizes two formats:

unsigned (corresponding to the C declaration unsigned int) -- all numbers are positive, a 1 in the most significant bit just means it is a really large number

signed (C declaration is signed int or just int)

-- numbers can be +/-, a 1 in the MSB means the number is negative

This distinction enables us to represent twice as many numbers when we're sure that we don't need negatives

#### Consider a comparison instruction: slt \$t0, \$t1, \$zero and \$t1 contains the 32-bit number 1111 01...01

What gets stored in \$t0?

Consider a comparison instruction: slt \$t0, \$t1, \$zero and \$t1 contains the 32-bit number 1111 01...01

What gets stored in \$t0? The result depends on whether \$t1 is a signed or unsigned number – the compiler/programmer must track this and accordingly use either slt or sltu

slt \$t0, \$t1, \$zero stores 1 in \$t0 sltu \$t0, \$t1, \$zero stores 0 in \$t0

- Occasionally, 16-bit signed numbers must be converted into 32-bit signed numbers – for example, when doing an add with an immediate operand
- The conversion is simple: take the most significant bit and use it to fill up the additional bits on the left – known as sign extension

and -2<sub>10</sub> goes from 1111 1111 1111 1110 to 1111 1111 1111 1111 1111 1110

- The following two (intuitive) representations were discarded because they required additional conversion steps before arithmetic could be performed on the numbers
  - sign-and-magnitude: the most significant bit represents
    +/- and the remaining bits express the magnitude
  - one's complement: -x is represented by inverting all the bits of x

Both representations above suffer from two zeroes

- Addition is similar to decimal arithmetic
- For subtraction, simply add the negative number hence, subtract A-B involves negating B's bits, adding 1 and A



Source: H&P textbook



- For an unsigned number, overflow happens when the last carry (1) cannot be accommodated
- For a signed number, overflow happens when the most significant bit is not the same as every bit to its left
  - when the sum of two positive numbers is a negative result
  - when the sum of two negative numbers is a positive result
  - The sum of a positive and negative number will never overflow
- MIPS allows addu and subu instructions that work with unsigned integers and never flag an overflow – to detect the overflow, other instructions will have to be executed

### **Multiplication Example**



In every step

- multiplicand is shifted
- next bit of multiplier is examined (also a shifting step)
- if this bit is 1, shifted multiplicand is added to the product

# HW Algorithm 1



In every step

- multiplicand is shifted
- next bit of multiplier is examined (also a shifting step)
- if this bit is 1, shifted multiplicand is added to the product

## HW Algorithm 2



Source: H&P textbook

- 32-bit ALU and multiplicand is untouched
- the sum keeps shifting right
- at every step, number of bits in product + multiplier = 64, hence, they share a single 64-bit register



- The previous algorithm also works for signed numbers (negative numbers in 2's complement form)
- We can also convert negative numbers to positive, multiply the magnitudes, and convert to negative if signs disagree
- The product of two 32-bit numbers can be a 64-bit number
  -- hence, in MIPS, the product is saved in two 32-bit registers

### **MIPS** Instructions

- mult\$s2, \$s3computes the product and stores<br/>it in two "internal" registers that<br/>can be referred to as hi and lo
- mfhi\$s0moves the value inhiinto\$s0mflo\$s1moves the value inlointo\$s1

Similarly for multu

# Fast Algorithm



- The previous algorithm requires a clock to ensure that the earlier addition has completed before shifting
- This algorithm can quickly set up most inputs – it then has to wait for the result of each add to propagate down – faster because no clock is involved

-- Note: high transistor cost

### Division



#### At every step,

- shift divisor right and compare it with current dividend
- if divisor is larger, shift 0 as the next bit of the quotient
- if divisor is smaller, subtract to get new dividend and shift 1 as the next bit of the quotient

#### Division



#### At every step,

- shift divisor right and compare it with current dividend
- if divisor is larger, shift 0 as the next bit of the quotient
- if divisor is smaller, subtract to get new dividend and shift 1 as the next bit of the quotient

#### **Divide Example**

• Divide  $7_{ten}$  (0000 0111<sub>two</sub>) by  $2_{ten}$  (0010<sub>two</sub>)

Iter	Step	Quot	Divisor	Remainder
0	Initial values			
1				
2				
3				
4				
5				

• Divide  $7_{ten} (0000 \ 0111_{two})$  by  $2_{ten} (0010_{two})$ 

lter	Step	Quot	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	Rem = Rem – Div	0000	0010 0000	1110 0111
	Rem < 0 ➔ +Div, shift 0 into Q	0000	0010 0000	0000 0111
	Shift Div right	0000	0001 0000	0000 0111
2	Same steps as 1	0000	0001 0000	1111 0111
		0000	0001 0000	0000 0111
		0000	0000 1000	0000 0111
3	Same steps as 1	0000	0000 0100	0000 0111
4	Rem = Rem – Div	0000	0000 0100	0000 0011
	Rem >= 0 ➔ shift 1 into Q	0001	0000 0100	0000 0011
	Shift Div right	0001	0000 0010	0000 0011
5	Same steps as 4	0011	0000 0001	0000 0001



#### Bullet